OpenAI Platform

Structured Outputs

Copy page

Ensure responses adhere to a JSON schema.

Try it out

Try it out in the Playground or generate a ready-to-use schema definition to experiment with structured outputs.

Generate

Introduction

JSON is one of the most widely used formats in the world for applications to exchange data.

Structured Outputs is a feature that ensures the model will always generate responses that adhere to your supplied JSON Schema, so you don't need to worry about the model omitting a required key, or hallucinating an invalid enum value.

Some benefits of Structured Outputs include:

1. **Reliable type-safety:** No need to validate or retry incorrectly formatted responses
2. **Explicit refusals:** Safety-based model refusals are now programmatically detectable
3. **Simpler prompting:** No need for strongly worded prompts to achieve consistent formatting

In addition to supporting JSON Schema in the REST API, the OpenAI SDKs for

Python and JavaScript also make it easy to define object schemas using Pydantic and Zod respectively. Below, you can see how to extract information from unstructured text that conforms to a schema defined in code.

Getting a structured response python

```python
from pydantic import BaseModel
from openai import OpenAI

client = OpenAI()

class CalendarEvent(BaseModel):
    name: str
    date: str
    participants: list[str]

completion = client.beta.chat.completions.parse(
    model="gpt-4o-2024-08-06",
    messages=[
        {"role": "system", "content": "Extract the event information."},
        {"role": "user", "content": "Alice and Bob are going to a scienc
    ],
    response_format=CalendarEvent,
)

event = completion.choices[0].message.parsed
```

Supported models

Structured Outputs is available in our latest large language models, starting with GPT-4o:

- `gpt-4.5-preview-2025-02-27` and later
- `o3-mini-2025-1-31` and later
- `o1-2024-12-17` and later
- `gpt-4o-mini-2024-07-18` and later

- `gpt-4o-2024-08-06` and later

Older models like `gpt-4-turbo` and earlier may use JSON mode instead.

When to use Structured Outputs via function calling vs via `response_format`

Structured Outputs is available in two forms in the OpenAI API:

1. When using function calling
2. When using a `json_schema` response format

Function calling is useful when you are building an application that bridges the models and functionality of your application.

For example, you can give the model access to functions that query a database in order to build an AI assistant that can help users with their orders, or functions that can interact with the UI.

Conversely, Structured Outputs via `response_format` are more suitable when you want to indicate a structured schema for use when the model responds to the user, rather than when the model calls a tool.

For example, if you are building a math tutoring application, you might want the assistant to respond to your user using a specific JSON Schema so that you can generate a UI that displays different parts of the model's output in distinct ways.

Put simply:

- If you are connecting the model to tools, functions, data, etc. in your system, then you should use function calling
- If you want to structure the model's output when it responds to the user, then you should use a structured `response_format`

The remainder of this guide will focus on non-function calling use cases in the Chat Completions API. To learn more about how to use Structured Outputs with function calling, check out the Function Calling guide.

Structured Outputs vs JSON mode

Structured Outputs is the evolution of JSON mode. While both ensure valid JSON is produced, only Structured Outputs ensure schema adherance. Both Structured Outputs and JSON mode are supported in the Chat Completions API, Assistants API, Fine-tuning API and Batch API.

We recommend always using Structured Outputs instead of JSON mode when possible.

However, Structured Outputs with `response_format: {type: "json_schema", ...}` is only supported with the `gpt-4o-mini`, `gpt-4o-mini-2024-07-18`, and `gpt-4o-2024-08-06` model snapshots and later.

	STRUCTURED OUTPUTS	JSON MODE
Outputs valid JSON	Yes	Yes
Adheres to schema	Yes (see supported schemas)	No
Compatible models	`gpt-4o-mini`, `gpt-4o-2024-08-06`, and later	`gpt-3.5-turbo`, `gpt-4-*` and `gpt-4o-*` models
Enabling	`response_format: { type: "json_schema", json_schema: {"strict": true, "schema": ...} }`	`response_format: { type: "json_object" }`

Examples

Chain of thought | Structured data extraction | UI generation | Moderation

Chain of thought

You can ask the model to output an answer in a structured, step-by-step way, to guide the user through the solution.

Structured Outputs for chain-of-thought math tutoring

python

```python
from pydantic import BaseModel
from openai import OpenAI

client = OpenAI()

class Step(BaseModel):
    explanation: str
    output: str

class MathReasoning(BaseModel):
    steps: list[Step]
    final_answer: str

completion = client.beta.chat.completions.parse(
    model="gpt-4o-2024-08-06",
    messages=[
        {"role": "system", "content": "You are a helpful math tutor. Gui
        {"role": "user", "content": "how can I solve 8x + 7 = -23"}
    ],
    response_format=MathReasoning,
)

math_reasoning = completion.choices[0].message.parsed
```

Example response

```
{
  "steps": [
    {
      "explanation": "Start with the equation 8x + 7 = -23.",
      "output": "8x + 7 = -23"
    },
    {
      "explanation": "Subtract 7 from both sides to isolate the term wit
      "output": "8x = -23 - 7"
    },
    {
      "explanation": "Simplify the right side of the equation.",
      "output": "8x = -30"
    },
    {
      "explanation": "Divide both sides by 8 to solve for x.",
      "output": "x = -30 / 8"
    },
    {
      "explanation": "Simplify the fraction.",
      "output": "x = -15 / 4"
    }
  ],
  "final_answer": "x = -15 / 4"
}
```


How to use Structured Outputs with `response_format`

You can use Structured Outputs with the new SDK helper to parse the model's output into your desired format, or you can specify the JSON schema directly.

Note: the first request you make with any schema will have additional latency as our API processes the schema, but subsequent requests with the same schema will not have additional latency.

SDK objects | Manual schema

Step 1: Define your object

Step 2: Supply your object in the API call

Step 3: Handle edge cases

Refusals with Structured Outputs

When using Structured Outputs with user-generated input, OpenAI models may occasionally refuse to fulfill the request for safety reasons. Since a refusal does not necessarily follow the schema you have supplied in `response_format`, the API response will include a new field called `refusal` to indicate that the model refused to fulfill the request.

When the `refusal` property appears in your output object, you might present the refusal in your UI, or include conditional logic in code that consumes the response to handle the case of a refused request.

python

```python
class Step(BaseModel):
    explanation: str
    output: str

class MathReasoning(BaseModel):
    steps: list[Step]
    final_answer: str

completion = client.beta.chat.completions.parse(
    model="gpt-4o-2024-08-06",
    messages=[
        {"role": "system", "content": "You are a helpful math tutor. Gui
        {"role": "user", "content": "how can I solve 8x + 7 = -23"}
    ],
    response_format=MathReasoning,
)

math_reasoning = completion.choices[0].message

# If the model refuses to respond, you will get a refusal message
if (math_reasoning.refusal):
    print(math_reasoning.refusal)
else:
    print(math_reasoning.parsed)
```

The API response from a refusal will look something like this:

```json
{
  "id": "chatcmpl-9nYAG9LPNonX8DAyrkwYfemr3C8HC",
  "object": "chat.completion",
  "created": 1721596428,
  "model": "gpt-4o-2024-08-06",
  "choices": [
    {
      "index": 0,
      "message": {
        "role": "assistant",
        "refusal": "I'm sorry, I cannot assist with that request."
      },
      "logprobs": null,
      "finish_reason": "stop"
    }
  ],
  "usage": {
    "prompt_tokens": 81,
    "completion_tokens": 11,
    "total_tokens": 92,
    "completion_tokens_details": {
      "reasoning_tokens": 0,
      "accepted_prediction_tokens": 0,
      "rejected_prediction_tokens": 0
    }
  },
  "system_fingerprint": "fp_3407719c7f"
}
```

Tips and best practices

Handling user-generated input

If your application is using user-generated input, make sure your prompt includes instructions on how to handle situations where the input cannot result in a valid response.

The model will always try to adhere to the provided schema, which can result in hallucinations if the input is completely unrelated to the schema.

You could include language in your prompt to specify that you want to return empty parameters, or a specific sentence, if the model detects that the input is incompatible with the task.

Handling mistakes

Structured Outputs can still contain mistakes. If you see mistakes, try adjusting your instructions, providing examples in the system instructions, or splitting tasks into simpler subtasks. Refer to the prompt engineering guide for more guidance on how to tweak your inputs.

Avoid JSON schema divergence

To prevent your JSON Schema and corresponding types in your programming language from diverging, we strongly recommend using the native Pydantic/zod sdk support.

If you prefer to specify the JSON schema directly, you could add CI rules that flag when either the JSON schema or underlying data objects are edited, or add a CI step that auto-generates the JSON Schema from type definitions (or vice-versa).

Streaming

You can use streaming to process model responses or function call arguments as they are being generated, and parse them as structured data.

That way, you don't have to wait for the entire response to complete before handling it. This is particularly useful if you would like to display JSON fields one by one, or handle function call arguments as soon as they are available.

We recommend relying on the SDKs to handle streaming with Structured Outputs. You can find an example of how to stream function call arguments without the SDK `stream` helper in the function calling guide.

Here is how you can stream a model response with the `stream` helper:

python

```python
from typing import List
from pydantic import BaseModel
from openai import OpenAI

class EntitiesModel(BaseModel):
    attributes: List[str]
    colors: List[str]
    animals: List[str]

client = OpenAI()

with client.beta.chat.completions.stream(
    model="gpt-4o",
    messages=[
        {"role": "system", "content": "Extract entities from the input t
        {
            "role": "user",
            "content": "The quick brown fox jumps over the lazy dog with
        },
    ],
    response_format=EntitiesModel,
) as stream:
    for event in stream:
        if event.type == "content.delta":
            if event.parsed is not None:
                # Print the parsed data as JSON
                print("content.delta parsed:", event.parsed)
        elif event.type == "content.done":
            print("content.done")
        elif event.type == "error":
            print("Error in stream:", event.error)

final_completion = stream.get_final_completion()
print("Final completion:", final_completion)
```

You can also use the `stream` helper to parse function call arguments:

python

```python
from pydantic import BaseModel
import openai
from openai import OpenAI

class GetWeather(BaseModel):
    city: str
    country: str

client = OpenAI()

with client.beta.chat.completions.stream(
    model="gpt-4o",
    messages=[
        {
            "role": "user",
            "content": "What's the weather like in SF and London?",
        },
    ],
    tools=[
        openai.pydantic_function_tool(GetWeather, name="get_weather"),
    ],
    parallel_tool_calls=True,
) as stream:
    for event in stream:
        if event.type == "tool_calls.function.arguments.delta" or event.
            print(event)

print(stream.get_final_completion())
```

Supported schemas

Structured Outputs supports a subset of the JSON Schema language.

Supported types

The following types are supported for Structured Outputs:

- String
- Number
- Boolean
- Integer
- Object
- Array
- Enum
- anyOf

Root objects must not be `anyOf`

Note that the root level object of a schema must be an object, and not use `anyOf`. A pattern that appears in Zod (as one example) is using a discriminated union, which produces an `anyOf` at the top level. So code such as the following won't work:

javascript

```javascript
import { z } from 'zod';
import { zodResponseFormat } from 'openai/helpers/zod';

const BaseResponseSchema = z.object({ /* ... */ });
const UnsuccessfulResponseSchema = z.object({ /* ... */ });

const finalSchema = z.discriminatedUnion('status', [
    BaseResponseSchema,
    UnsuccessfulResponseSchema,
]);

// Invalid JSON Schema for Structured Outputs
const json = zodResponseFormat(finalSchema, 'final_schema');
```

All fields must be `required`

To use Structured Outputs, all fields or function parameters must be specified as `required`.

```json
{
    "name": "get_weather",
    "description": "Fetches the weather in the given location",
    "strict": true,
    "parameters": {
        "type": "object",
        "properties": {
            "location": {
                "type": "string",
                "description": "The location to get the weather for"
            },
            "unit": {
                "type": "string",
                "description": "The unit to return the temperature in",
                "enum": ["F", "C"]
            }
        },
        "additionalProperties": false,
        "required": ["location", "unit"]
    }
}
```

Although all fields must be required (and the model will return a value for each parameter), it is possible to emulate an optional parameter by using a union type with `null`.

```json
{
    "name": "get_weather",
    "description": "Fetches the weather in the given location",
    "strict": true,
    "parameters": {
        "type": "object",
        "properties": {
            "location": {
                "type": "string",
                "description": "The location to get the weather for"
            },
            "unit": {
                "type": ["string", "null"],
                "description": "The unit to return the temperature in",
                "enum": ["F", "C"]
            }
        },
        "additionalProperties": false,
        "required": [
            "location", "unit"
        ]
    }
}
```

Objects have limitations on nesting depth and size

A schema may have up to 100 object properties total, with up to 5 levels of nesting.

Limitations on total string size

In a schema, total string length of all property names, definition names, enum values, and const values cannot exceed 15,000 characters.

Limitations on enum size

A schema may have up to 500 enum values across all enum properties.

For a single enum property with string values, the total string length of all enum values cannot exceed 7,500 characters when there are more than 250 enum values.

`additionalProperties: false` must always be set in objects

`additionalProperties` controls whether it is allowable for an object to contain additional keys / values that were not defined in the JSON Schema.

Structured Outputs only supports generating specified keys / values, so we require developers to set `additionalProperties: false` to opt into Structured Outputs.

json

```
{
    "name": "get_weather",
    "description": "Fetches the weather in the given location",
    "strict": true,
    "schema": {
        "type": "object",
        "properties": {
            "location": {
                "type": "string",
                "description": "The location to get the weather for"
            },
            "unit": {
                "type": "string",
                "description": "The unit to return the temperature in",
                "enum": ["F", "C"]
            }
        },
        "additionalProperties": false,
        "required": [
            "location", "unit"
        ]
    }
}
```

Key ordering

When using Structured Outputs, outputs will be produced in the same order as the ordering of keys in the schema.

Some type-specific keywords are not yet supported

Notable keywords not supported include:

- **For strings:** `minLength`, `maxLength`, `pattern`, `format`
- **For numbers:** `minimum`, `maximum`, `multipleOf`
- **For objects:** `patternProperties`, `unevaluatedProperties`, `propertyNames`, `minProperties`, `maxProperties`
- **For arrays:** `unevaluatedItems`, `contains`, `minContains`, `maxContains`, `minItems`, `maxItems`, `uniqueItems`

If you turn on Structured Outputs by supplying `strict: true` and call the API with an unsupported JSON Schema, you will receive an error.

For `anyOf`, the nested schemas must each be a valid JSON Schema per this subset

Here's an example supported anyOf schema:

json

```
{
    "type": "object",
    "properties": {
        "item": {
            "anyOf": [
                {
                    "type": "object",
                    "description": "The user object to insert into the d
                    "properties": {
                        "name": {
                            "type": "string",
                            "description": "The name of the user"
                        },
```

```
                        "age": {
                            "type": "number",
                            "description": "The age of the user"
                        }
                    },
                    "additionalProperties": false,
                    "required": [
                        "name",
                        "age"
                    ]
                },
                {
                    "type": "object",
                    "description": "The address object to insert into th
                    "properties": {
                        "number": {
                            "type": "string",
                            "description": "The number of the address. E
                        },
                        "street": {
                            "type": "string",
                            "description": "The street name. Eg. for 123
                        },
                        "city": {
                            "type": "string",
                            "description": "The city of the address"
                        }
                    },
                    "additionalProperties": false,
                    "required": [
                        "number",
                        "street",
                        "city"
                    ]
                }
            ]
        }
    },
    "additionalProperties": false,
    "required": [
        "item"
    ]
```

```
56 }
```

Definitions are supported

You can use definitions to define subschemas which are referenced throughout your schema. The following is a simple example.

json

```json
{
    "type": "object",
    "properties": {
        "steps": {
            "type": "array",
            "items": {
                "$ref": "#/$defs/step"
            }
        },
        "final_answer": {
            "type": "string"
        }
    },
    "$defs": {
        "step": {
            "type": "object",
            "properties": {
                "explanation": {
                    "type": "string"
                },
                "output": {
                    "type": "string"
                }
            },
            "required": [
                "explanation",
                "output"
            ],
            "additionalProperties": false
        }
    },
    "required": [
        "steps",
        "final_answer"
    ],
    "additionalProperties": false
}
```

Recursive schemas are supported

Sample recursive schema using `#` to indicate root recursion.

```json
{
    "name": "ui",
    "description": "Dynamically generated UI",
    "strict": true,
    "schema": {
        "type": "object",
        "properties": {
            "type": {
                "type": "string",
                "description": "The type of the UI component",
                "enum": ["div", "button", "header", "section", "field",
            },
            "label": {
                "type": "string",
                "description": "The label of the UI component, used for
            },
            "children": {
                "type": "array",
                "description": "Nested UI components",
                "items": {
                    "$ref": "#"
                }
            },
            "attributes": {
                "type": "array",
                "description": "Arbitrary attributes for the UI componen
                "items": {
                    "type": "object",
                    "properties": {
                        "name": {
                            "type": "string",
                            "description": "The name of the attribute, f
                        },
                        "value": {
                            "type": "string",
                            "description": "The value of the attribute"
                        }
                    },
```

```
                    "additionalProperties": false,
                    "required": ["name", "value"]
                }
            }
        },
        "required": ["type", "label", "children", "attributes"],
        "additionalProperties": false
    }
}
```

Sample recursive schema using explicit recursion:

json

```json
{
    "type": "object",
    "properties": {
        "linked_list": {
            "$ref": "#/$defs/linked_list_node"
        }
    },
    "$defs": {
        "linked_list_node": {
            "type": "object",
            "properties": {
                "value": {
                    "type": "number"
                },
                "next": {
                    "anyOf": [
                        {
                            "$ref": "#/$defs/linked_list_node"
                        },
                        {
                            "type": "null"
                        }
                    ]
                }
            },
            "additionalProperties": false,
            "required": [
                "next",
                "value"
            ]
        }
    },
    "additionalProperties": false,
    "required": [
        "linked_list"
    ]
}
```

JSON mode

JSON mode is a more basic version of the Structured Outputs feature. While JSON mode ensures that model output is valid JSON, Structured Outputs reliably matches the model's output to the schema you specify. We recommend you use Structured Outputs if it is supported for your use case.

When JSON mode is turned on, the model's output is ensured to be valid JSON, except for in some edge cases that you should detect and handle appropriately.

To turn on JSON mode with the Chat Completions or Assistants API you can set the `response_format` to `{ "type": "json_object" }`. If you are using function calling, JSON mode is always turned on.

Important notes:

- When using JSON mode, you must always instruct the model to produce JSON via some message in the conversation, for example via your system message. If you don't include an explicit instruction to generate JSON, the model may generate an unending stream of whitespace and the request may run continually until it reaches the token limit. To help ensure you don't forget, the API will throw an error if the string "JSON" does not appear somewhere in the context.
- JSON mode will not guarantee the output matches any specific schema, only that it is valid and parses without errors. You should use Structured Outputs to ensure it matches your schema, or if that is not possible, you should use a validation library and potentially retries to ensure that the output matches your desired schema.
- Your application must detect and handle the edge cases that can result in the model output not being a complete JSON object (see below)

Handling edge cases

Resources

To learn more about Structured Outputs, we recommend browsing the following resources:

- Check out our introductory cookbook on Structured Outputs
- Learn how to build multi-agent systems with Structured Outputs